Diviser pour régner

L'idée de départ :

Nous avons observé que les algorithmes récursifs étaient souvent gourmands en complexité sauf dans les deux cas suivants dont la complexité est logarithmique :

1. Algorithme d'exponentiation rapide
2. Algorithme de recherche dichotomique dans un tableau trié

La particularité de ces deux exemples est que l'unique appel récursif porte sur un argument dont la taille est deux fois moins grande que l'argument initial. C'est l'idée des algorithmes de type "diviser pour régner" : faire en sorte que le ou les appels récursifs portent sur un argument dont la taille est une fraction de l'argument initial.

I] Exemples d'algorithmes usuels de type « Diviser Pour Régner »

Exemple 1 : Exponentiation rapide

OCaml
```
let rec f a = function
            | 0 -> 1
            | n -> match n mod 2 with
                    | 0 -> let c = f a (n/2) in c*c        (* 1 appel récursif *)
                    | 1 -> let c = f a (n/2) in a*c*c;;    (* 1 appel récursif *)
```

La taille (n) de l'argument est divisée par 2.

Exemple 2 : Recherche dichotomique

OCaml
```
let recherche e t = let n = Array.length t in
                    let rec dico e t a b =
                       match b-a <= 1 with                          (* condition d'arrêt *)
                        | true  -> e = t.(a) || e = t.(b)
                        | false ->  if t.((a+b)/2) = e
                                    then true
                                    else match t.((a+b)/2) < e with
                                         | true  -> dico e t ((a+b)/2) b      (* 1 appel *)
                                         | false -> dico e t a ((a+b)/2) in   (* 1 appel *)
                    dico e t 0 (n-1) ;;
```

La taille $(b-a)$ de l'argument est divisée par 2.

Exemple 3 : Le tri fusion

OCaml
```
let rec divise = function
            | [] -> [],[]
            | [x] -> [x],[]
            | x::y::q -> let l1,l2 = divise q in x::l1,y::l2;;          (* 1 appel*)

divise [1;3;7;8;9];;

let rec fusion = function
          | [],[]  -> []
          | l,[] -> l
          | [],l -> l
          | x1::q1,x2::q2 -> match x1 < x2 with
                              | true  -> x1::(fusion (q1,(x2::q2)))     (* 1 appel*)
                              | false -> x2::(fusion ((x1::q1),q2));;   (* 1 appel*)

let rec tri = function
          | [] -> []
          | [x] -> [x]
          | l   -> let l1,l2 = divise l in
                   fusion ((tri l1),(tri l2));;                         (* 2 appels*)

tri [7;8;3;4;8;2];;
```

La fonction `tri` possède 2 appels récursifs portant l'un et l'autre sur un argument deux fois moins grand que l'argument initial.

Exemple 4 : Algorithme de Strassen

Le principe de cet algorithme est le suivant :

Pour multiplier deux matrices M et N de $\mathfrak{M}_n(\mathbb{R})$:

- on commence par décomposer les 2 matrices en 4 blocs.

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \quad \text{et} \quad N = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

- On calcule alors récursivement les matrices suivantes :

$$\begin{array}{lll} P_1 = A(F-H) & P_4 = D(G-E) & P_7 = (A-C)(E+F) \\ P_2 = (A+B)H & P_5 = (A+D)(E+H) & \\ P_3 = (C+D)E & P_6 = (B-D)(G+H) & \end{array}$$

- On remarque alors que : $MN = \begin{pmatrix} -P_2+P_4+P_5+P_6 & P_1+P_2 \\ P_3+P_4 & P_1-P_3+P_5-P_7 \end{pmatrix}$

Dans cet algorithme, nous avons 7 appels récursifs portant sur des arguments dont la taille a été divisée par 2 par rapport à la taille des arguments initiaux.

Exemple 5 : Karatsuba pour la multiplication des entiers

Le principe de cet algorithme est le suivant :

- On note $\begin{cases} p \\ q \end{cases}$ les deux entiers à multiplier, n le nombre de chiffres du plus grand des deux et $m = \lfloor \frac{n}{2} \rfloor$.

  On décompose p le plus grand des deux nombres (grossièrement) :

$$\begin{aligned} p &= a_n 10^n + \cdots + a_{\frac{n}{2}} 10^{\frac{n}{2}} + a_1 10 + a_0 \\ &= \left(a_n 10^{\frac{n}{2}} + \cdots + a_{\frac{n}{2}}\right) 10^{\frac{n}{2}} + (a_{\frac{n}{2}-1} 10^{\frac{n}{2}-1} + \ldots a_1 10 + a_0) \\ &= a.10^{\frac{n}{2}} + b \end{aligned}$$

- On a alors p et q de la forme $\begin{cases} p = a.10^m + b \\ q = c.10^m + d \end{cases}$ où le nombre de chiffres de $a,\ b,\ c,\ d$ est au plus $m+1$.
- Remarquons alors que : $pq = (a.10^m + b)(c.10^m + d) = ac.10^{2m} + (ad + cb).10^m + bd$.

  En posant $\begin{cases} x_1 = ac \\ x_2 = (a+b)(c+d) \\ x_3 = bd \end{cases}$, on vérifie que $\begin{cases} ac = x_1 \\ ad + cb = x_2 - x_1 - x_3 \\ bd = x_3 \end{cases}$ et nous avons donc :

$$\boxed{pq = x_1 10^{2m} + (x_2 - x_1 - x_3)10^m + x_3}$$

Dans cet algorithme, nous avons 3 appels récursifs portant sur des arguments dont la taille a été divisée par 2 par rapport à la taille des arguments initiaux.

Exemples vus en partie III] et IV]

- Calcul de la plus petite distance dans un nuage de points
- Calcul du nombre d'inversions dans une permutation

II] Etude de la complexité

On constate que la complexité $c(n)$ d'un algorithme de type Diviver Pour Régner vérifie en général une relation de récurrence de la forme :

$$c(n) = a.c(\frac{n}{b}) + f(n) \quad \text{avec} \quad \begin{cases} a \text{ le nombre d'appels récursifs} \\ b \text{ la valeur qui divise la taille de l'argument} \\ f(n) \text{ le coût des traitements complémentaires} \end{cases}$$

Le problème : Comment obtenir un ordre de grandeur de la complexité $c(n)$ vérifiant une telle relation ?

Pour simplifier un peu l'exposé qui suit, nous allons nous placer dans le cas où $b = 2$:

$$c(n) = a.c(\frac{n}{2}) + f(n)$$

Structure de la méthode :

$$c(n) = a.c(\frac{n}{2}) + f(n)$$

- **Etape 1** : on suppose que n est une puissance de 2 ($n = 2^k$) et on détermine $c(2^k) = O(u(k))$
- **Etape 2** : on généralise au cas où n est quelconque

  → en encadrant : $2^{k-1} \leq n \leq 2^k$

  → en admettant alors que : $c(n) \leq c(2^k)$

  La connaissance de l'ordre de grandeur de $c(2^k)$ nous permet alors de conclure...

Etape 1 : On suppose que n est une puissance de 2 : $n = 2^k$

- On a alors : $c(2^k) = a.c(2^{k-1}) + f(2^k)$ et donc en divisant par a^k : $\frac{c(2^k)}{a^k} = \frac{c(2^{k-1})}{a^{k-1}} + \frac{f(2^k)}{a^k}$.
- On pose $u_k = \frac{c(2^k)}{a^k}$ qui vérifie donc : $u_k = u_{k-1} + \frac{f(2^k)}{a^k}$.
- Nous obtenons alors : $u_n = u_0 + \sum_{k=1}^{n} \frac{f(2^k)}{a^k}$ et donc :

  → 1er cas : $\sum_{k=1}^{n} \frac{f(2^k)}{a^k} = O(1)$ et donc $u_n = O(1)$ soit $\frac{c(2^n)}{a^n} = O(1)$.

  On a alors : $\boxed{c(2^n) = O(a^n)}$.

  → 2eme cas : $\sum_{k=1}^{n} \frac{f(2^k)}{a^k} \xrightarrow[n \to +\infty]{} +\infty$.

  Dans ce cas, nous avons $u_n = u_0 + \sum_{k=1}^{n} \frac{f(2^k)}{a^k} \sim \sum_{k=1}^{n} \frac{f(2^k)}{a^k} = O(\sum_{k=1}^{n} \frac{f(2^k)}{a^k})$.

  Et donc :

$$\boxed{c(2^n) = O\left(a^n . \sum_{k=1}^{n} \frac{f(2^k)}{a^k}\right)}$$

Etape 2 : On généralise au cas où n est quelconque

On considère k tel que : $2^{k-1} \leq n \leq 2^k$.

- On a alors : $k = O(\ln(n))$ et $2^k = O(n)$

Preuve

→ Comme $2^{k-1} \leq n \leq 2^k \Rightarrow (k-1)\ln 2 \leq \ln n \leq k \ln 2 \Rightarrow k - 1 \leq \frac{\ln n}{\ln 2} \leq k$.

Nous avons ainsi $\dfrac{\ln n}{\ln 2} \leq k \leq \dfrac{\ln n}{\ln 2} + 1$ et donc $\boxed{k = O(\ln(n))}$

→ Comme $2^{k-1} \leq n \leq 2^k$ alors $n \leq 2^k \leq 2n$ et donc $\boxed{2^k = O(n)}$

- On admet que la complexité est croissante :

Nous avons alors $c(n) \leq c(2^k)$ et nous pouvons alors utiliser les résultats précédents.

Voyons ce que cela donne dans les deux cas usuels suivants :

Cas d'un seul appel récursif ($a = 1$)

$$c(n) \leq c(2^k) \quad \text{avec} \quad c(2^k) = O\left(\sum_{i=1}^{k} f(2^i)\right)$$

- Lorsque $f(n) = O(1)$:

→ Nous avons $0 \leq \sum_{i=1}^{k} f(2^i) \leq \sum_{i=1}^{k} M = kM$ donc $c(2^k) = O(k)$.

→ Nous avons donc $c(n) \leq O(k)$ et comme $k = O(\ln(n))$ on en déduit que $\boxed{c(n) = O(\ln n)}$

- Lorsque $f(n) = O(n)$:

→ Nous avons $0 \leq \sum_{i=1}^{k} f(2^i) \leq \sum_{i=1}^{k} M2^i = 2M\dfrac{1-2^k}{1-2} = O(2^k)$ donc $c(2^k) = O(2^k)$.

→ Nous avons donc $c(n) \leq O(2^k)$ et comme $2^k = O(n)$ on en déduit que $\boxed{c(n) = O(n)}$

Cas de 2 appels récursifs ($a = 2$)

$$c(n) \leq c(2^k) \quad \text{avec} \quad c(2^k) = O\left(2^k . \sum_{i=1}^{k} \frac{f(2^i)}{2^i}\right)$$

- Lorsque $f(n) = O(1)$:

  → Nous avons $0 \leq 2^k . \sum_{i=1}^{k} \frac{f(2^i)}{2^i} \leq 2^k \sum_{i=1}^{k} \frac{M}{2^i} = 2^k M'$ donc $\quad c(2^k) = O(2^k)$.

  → Nous avons donc $c(n) \leq O(2^k)$ et comme $2^k = O(n)$ on en déduit que $\boxed{c(n) = O(n)}$

- Lorsque $f(n) = O(n)$:

  → Nous avons $0 \leq 2^k . \sum_{i=1}^{k} \frac{f(2^i)}{2^i} \leq 2^k . \sum_{i=1}^{k} M = 2^k . Mk$ donc $\quad c(2^k) = O(k.2^k)$.

  → Nous avons donc $c(n) \leq O(k2^k)$ et comme $\left\{ \begin{array}{l} k = O(\ln(n)) \\ 2^k = O(n) \end{array} \right.$ on en déduit que $\boxed{c(n) = O(n \ln n)}$

Nombre d'appels récursifs	1 appel	1 appel	2 appels	2 appels
Traitements complémentaires	$f(n) = O(1)$	$f(n) = O(n)$	$f(n) = O(1)$	$f(n) = O(n)$
Complexité de l'algorithme	$O(\ln n)$	$O(n)$	$O(n)$	$O(n \ln n)$

Les complexités obtenues sont très satisfaisantes ! !

L'ordre de grandeur est à connaître, mais les valeurs de ces complexités ne doivent pas être apprises par coeur. Il faut être capable de refaire les calculs ! !

Exemple 1 : Complexité de l'exponentiation rapide

OCaml
```
let rec f a = function
            | 0 -> 1
            | n -> match n mod 2 with
                    | 0 -> let c = f a (\frac n2) in c*c        (* 1 appel récursif *)
                    | 1 -> let c = f a (\frac n2) in a*c*c;;    (* 1 appel récursif *)
```

Cette complexité vérifie la relation $c(n) = c(\frac{n}{2}) + O(1)$.

Nous sommes donc dans le cas où $\left\{ \begin{array}{l} a = 1 \\ f(n) = O(1) \end{array} \right.$ et donc : $\boxed{c(n) = O(\ln(n))}$.

Exemple 2 : Complexité du tri fusion

OCaml
```
let rec tri = function
          | [] -> []
          | [x] -> [x]
          | l   -> let l1,l2 = divise l in
                   fusion ((tri l1),(tri l2));;                    (* 2 appels*)
```

Question : Montrer que la complexité vérifie la relation $c(n) = 2c(\frac{n}{2}) + O(n)$.

Nous sommes donc dans le cas où $\begin{cases} a = 2 \\ f(n) = O(n) \end{cases}$ et donc : $\boxed{c(n) = O(n\ln(n))}$.

Exemple 3 : Complexité de Karatsuba

Pour multiplier deux entiers p et q :

- Algorithme : On écrit $\begin{cases} p = a.10^m + b \\ q = c.10^m + d \end{cases}$ avec m la longueur du plus grand nombre divisée par 2.
  En posant $\begin{cases} x_1 = ac \\ x_2 = (a+b)(c+d) \\ x_3 = bd \end{cases}$, on vérifie que $\boxed{pq = x_1 10^{2m} + (x_2 - x_1 - x_3)10^m + x_3}$.
- Taille : n le nombre de chiffres du plus grand entier.
- Unité de coût : la multiplication.
- Complexité : La complexité vérifie la relation : $c(n) = 3c(\frac{n}{2}) + 6$

  → Etape 1 : Prenons $n = 2^k$.

  $c(2^k) = 3c(2^{k-1}) + 6$ et donc $\frac{c(2^k)}{3^k} = \frac{c(2^{k-1})}{3^{k-1}} + \frac{6}{3^k}$ qui donne en sommant :

  $$\frac{c(2^n)}{3^n} = c(1) + \sum_{k=1}^{n} \frac{6}{3^k} = c(1) + 2.\frac{1 - (\frac{1}{3})^n}{1 - \frac{1}{3}} = O(1) \quad \text{et donc} \quad \boxed{c(2^n) = O(3^n)}$$

  → Etape 2 : Soit $n \in \mathbb{N}$.

  On introduit k tel que $2^{k-1} \leq n \leq 2^k$ et nous avons alors : $k \leq \frac{\ln n}{\ln 2} + 1$.
  Ainsi :

  $$c(n) \leq c(2^k) = O(3^k) \leq M.3^k \leq 3M3^{\frac{\ln n}{\ln 2}} \leq M'.e^{\frac{\ln n}{\ln 2}\ln 3} \leq M'.e^{\ln n^{\frac{\ln 3}{\ln 2}}} \leq M'.n^{\frac{\ln 3}{\ln 2}}$$

  → Conclusion :

  $$\boxed{c(n) = O(n^{\frac{\ln 3}{\ln 2}}) \equiv O(n^{1.58\ldots})}$$

Exercice : 1

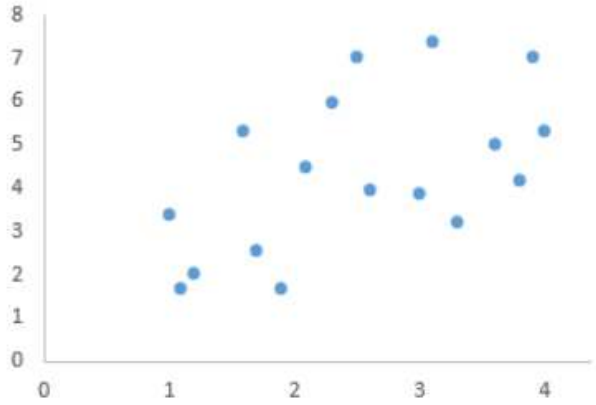
Etude de la complexité de l'algorithme de Strassen

1. Que vaut la complexité (en nombre d'additions élémentaires) d'un algorithme naïf de produit matriciel de 2 matrices de taille nxn ?
2. Que vaut la complexité de l'algorithme de Strassen pour produit matriciel de 2 matrices de taille nxn ?
3. Comparer ces 2 complexités !

III] Calcul de la plus petite distance dans un nuage de points

Il s'agit ici de déterminer la plus petite distance entre deux points contenus dans un nuage de points de $\mathbb{R}^2$.
ce nuage est donné sous la forme d'un tableau t de points eux-même représentés par des tableaux à 2 composantes.

```
t = [|[|4;8|];[|-1;5|];[|0;-2|];[|4;1|];[|3;-1|];[|3;0|];[|1;6|]|]
```



A] Avec un algorithme naïf

- Principe : On calcule toutes les distances possibles à l'aide de 2 « boucles for » imbriquées.
- Complexité : Pour un tableau de taille n

$$c_1(n) = (n-1) + (n-2) + \cdots + 1 = O(n^2)$$

B] Avec un algorithme de type « Diviser pour Régner »

1) Principe de l'algorithme

Etape 1 : On commence par trier ce nuage par ordre croissant :

- selon leurs abscisses pour obtenir un tableau

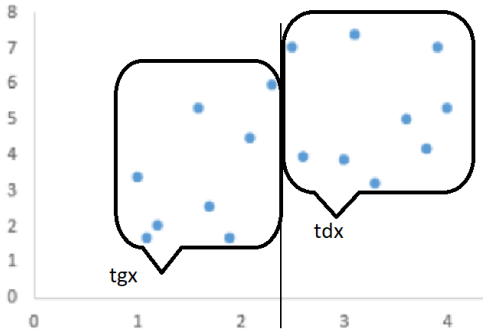
```
tx = [|[|-1;5|];[|0;-2|];[|1;6|];[|3;-1|];[|3;0|];[|4;1|]|]
```

- selon leurs ordonnées pour obtenir un tableau

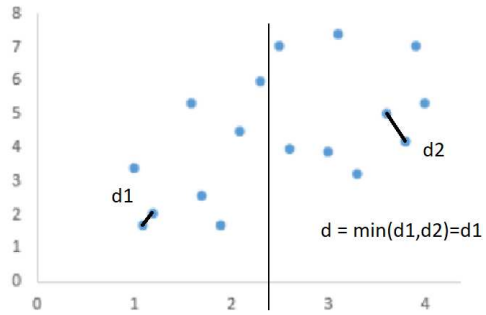
```
ty = [|[|0;-2|];[|3;-1|];[3;0|];[|4;1|];[|-1;5|];[|1;6|]|]
```

Etape 2 :

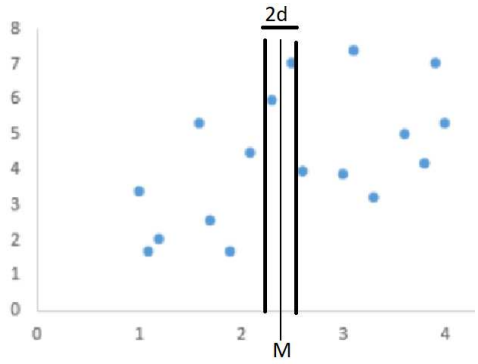
- On détermine à partir de `tx` la médiane M des abscisses (algorithme de complexité linéaire).
- On répartit les points de `tx` en deux sous-nuages triés selon les abscisses :
  → `tgx` contenant les points d'abscisse inférieure à M
  → `tdx` contenant les autres points.



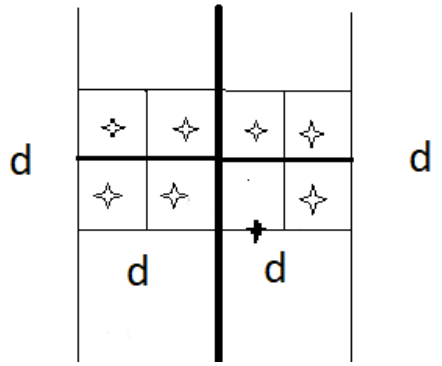
Etape 3 :
On applique récursivement la fonction aux deux de nuages `tgx` et `tgx`.
On appelle d le minimum des deux distances minimales obtenues.



Etape 4 :
On considère alors un troisième nuage `tmil` contenant par ordre croissant selon les ordonnées, tous les points d'abscisse comprise entre $M - d$ et $M + d$ (il suffit pour cela d'éliminer les points du nuage `ty`) qui ne vérifient pas cette condition et on regarde si le nouveau nuage contient deux points dont la distance est inférieure à d.



Les points de ce tableau `tmil` étant classés selon leur ordonnée, on remarque qu'il suffit de ne considérer que les distances entre un point et ses 7 successeurs dans la liste.



En effet, si l'on considère une tranche de hauteur d de `tmil` à partir du point considéré et qu'on la découpe en 8 cases de côté $\frac{d}{2}$, on remarque que :

→ les points qui ne sont pas dans cette tranche sont à une distance supérieure à d de celui-ci

→ chaque case de cette tranche ne peut contenir qu'un seul point.

2) Définition des différentes fonctions nécessaires :

Dans cet algorithme, nous avons besoin :

- D'une fonction permettant de trier par ordre croissant un tableau de points selon les abscisses ou les ordonnées.

OCaml
```
Nom : tri_tab
Arguments - t : tableau de points
          - a : 0 si on veut trier par rapport à x
                1 si on veut trier par rapport à y
Sortie : le tableau t trié selon les abscisses ou les ordonnées
Algorithme : tri fusion ou tri rapide
Complexité : O(n.ln(n))
```

- D'une fonction permettant de déterminer la médiane des abscisses des points d'un tableau trié.

OCaml
```
Nom : mediane
Argument - t ;: tableau de points trié selon les abscisses
Sortie : la valeur médiane des abscisses
Algorithme : On renvoie la moyenne entre les abscisses des points t.(n/2) et t.(n/2+1)
Complexité : O(1)
```

- D'une fonction qui décompose un tableau en deux sous-tableaux selon la valeur des abscisses par rapport à la médiane

OCaml
```
Nom : decompose
Arguments - t : tableau trié selon les abscisses
Sortie : deux tableaux de même taille triés par ordre croissant
          - l'un avec les n/2 premiers points
          - l'autre avec les n/2 suivants
Algorithme : basique
Complexité : O(n)
```

- D'une fonction calculant la distance entre deux points.

OCaml
```
Nom : dist
Argument - A un point représenté par un tableau de longueur 2
         - B un autre proint
Sortie : La distance AB
Algorithme : Application de la formule usuelle
Comlexité : O(1)
```

- D'une fontion qui ne sélectionne que les points d'un tableau dont les abscisses sont comprises entre 2 valeurs.

OCaml
```
Nom : t_centre
Arguments - t : tableau de points trié selon les ordonnées
          - m : médiane des abscisses
          - d : distance de part et d'autre m
Sortie : un nouveau tableau ne contenant que les points de t souhaités
Algorithme : Basique
Complexité : O(n)
```

- D'une fonction qui détermine la plus petite distance entre 2 points du tableau dont les indices vérifient $|j-i| \leq 7$.

OCaml
```
Nom : dist_tcentre
Argument : t un tableau de points trié selon les ordonnées
Sortie : la distance minimale entre points d'indices vérifiant |j-i|<=7
Algorithme : pour chaque point, on compare sa distance
             aux 7 points suivants à la distance minimale stockée
Complexité : O(n)
```

- De la fonction globale de calcul de la plus petite distance.

OCaml
```
let rec distmin t = let tx = tri_tab t 0 and        (* On ordonne le tableau *)
                        ty = tri_tab t 1 in
                    let m = mediane tx in           (* médiane des abscisses *)
                    let tgx,tdx = decompose tx in   (* décomposition du nuage en deux nuage *)
                    let dg = distmin tgx and        (* distances minimales dans chacun... *)
                        dd = distmin tdx in         (* ...des deux sous-nuages *)
                    let d = min dg dd in            (* minimum des distances précédentes *)
                    let tc = t_centre ty m d in     (* tableau central *)
                    let dc = dist_tcentre tc in     (* distance minimale dans le tableau central *)
                    min d dc;;                      (* on renvoie la distance minimale globale *)
```

3) Calcul de la complexité :

- Taille : n la longueur du tableau
- Unité de coût : ça dépend des fonctions...
- Complexité :

  → Nous avons 2 appels récursifs sur des arguments de taille $\frac{n}{2}$

  → Traitements complémentaires :

  ⋆ Calcul de `tx` et de `ty` : `2.O(n.ln(n))` avec un tri rapide ou un tri fusion

  ⋆ Calcul de la médiane : `O(1)`

  ⋆ Décomposition en 2 nuages : `O(n)`

  ⋆ Calcul de d : `O(1)`

- ⋆ Détermination du tableau central : `O(n)`
- ⋆ Distance minimale dans le tableau central : `O(n)`

$c(n)$ vérifie donc la relation de récurrence

$$c(n) = 2c(\frac{n}{2}) + 2.O(n\ln(n)) + O(1) + O(n) + O(1) + O(n) + O(n) \quad \text{soit} \quad \boxed{c(n) = 2c(\frac{n}{2}) + O(n\ln n)}$$

Questions

1. Montrer que la complexité de notre algorithme est un $O(n\ln^2(n)$
2. Comparer cette complexité à celle de l'algorithme naïf utilisant la force brute

IV] Calcul du nombre d'inversions

Il s'agit ici de déterminer le nombre d'inversions contenues dans un tableau contenant n entiers distincts.
Cet algorithme est en particulier utilisé dans les cas suivants :

1. Comparaison des goûts de deux personnes ayant classés des objets par ordre de préférence
2. Mesure du degré de rangement d'un tableau
3. Analyse de la sensibilité du ranking de google

A] Avec un algorithme naïf

- Principe :
  → On parcourt l'ensemble des couples `(t.(i), t.(j)) pour i < j`
  → on compte le nombre de fois où l'on a `t.(i) > t.(j)`

- Complexité : Pour un tableau de taille n

$$c_1(n) = (n-1) + (n-2) + \cdots + 1 = O(n^2)$$

B] Avec un algorithme de type « Diviser pour Régner »

1) Principe général de l'algorithme

- On sépare le tableau en deux tableaux de taille égale (ou presque) en gardant l'ordre initial des éléments.

OCaml
```
Nom : decompose
Argument - t : un tableau d'entier
Sortie : t1 et t2 deux sous-tableaux de taille égale (à 1 près)
Algorithme : Basique
Complexité : O(n)
```

- On applique récursivement la fonction pour compter le nombre d'inversions dans chacun des deux tableaux.

- Il reste alors à déterminer le nombre d'inversions du type $(i,\ j)$ où i est un élément du premier tableau et j un élément du deuxième.

  → Pour cela, on pourrait comparer tous les couples possibles mais cela donnerait $\frac{n^2}{4}$ comparaisons et cela remettrait en cause la pertinence de notre algorithme.

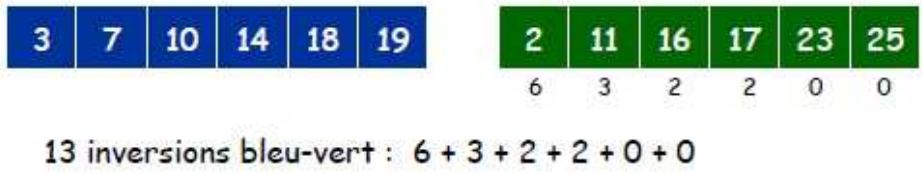
  → On décide plutôt de :
  — commencer par ordonner les éléments de chacun des tableaux (complexité en $O(n \ln n)$),
  — puis on détermine le nombre d'inversions à l'aide de l'algorithme ci-dessous.

Dénombrement des inversions entre les 2 sous-tableaux triés

- On note n_1 la longueur du premier tableau `t1` et n_2 la longueur de tableau `t2`.
- On souhaite compter le nombre d'inversions de type $(i,\ j)$ où $i \in [\![0, n_1 - 1]\!]$ et $j \in [\![0, n_2 - 1]\!]$.
- Pour j allant de 0 à $n_2 - 1$:

  → On détermine la première valeur de i telle que `t1.(i) > t2.(j)` qui donne la première inversion. Dans ce cas, le tableau `t1` étant trié, tous les couples $(k,\ j)$ tels que $k > i$ seront aussi des inversions. Nous aurons alors déterminé $n_1 - i$ inversions du type $(x,\ j)$, nombre à ajouter à un compteur S.

  → Le tableau `t2` étant trié, on cherchera la valeur de i en partant de la valeur i précédemment trouvée car si $(x,\ j)$ n'était pas une inversion, alors $(x,\ j+1)$ ne le sera pas non plus.



Application à l'exemple ci-dessus

Décompte du nombre d'inversions :

- Pour $j = 0$:
  $(i = 0, j = 0)$ est une inversion et donc $(i, j = 0)$ sera une inversion pour tout $i \geq 0$: 6 inversions
- Pour $j = 1$:
  La première inversion est obtenue pour $(i = 3, j = 1)$.
  Les couples $(i, j = 1)$ seront donc des inversions pour tout $i \geq 3$: 3 inversions
- Pour $j = 2$:
  Inutile de tester les couples de la forme (i, j) avec $i < 3$ car ils ne peuvent donner des inversions.
  La première inversion est obtenue pour $(i = 4, j = 1)$.
  Les couples $(i, j = 1)$ seront donc des inversions pour tout $i \geq 4$: 2 inversions
- Pour $j = 3$:
  On ne teste que les couples $(i, j = 3)$ avec $i \geq 4$ car les autres ne peuvent donner des inversions.
  On constate qu'il n'y a plus d'inversion.

Le nombre total d'inversions est donc $6 + 3 + 2 + 0 + 0 = 11$ inversions

Décompte du nombre de tests effectués :

→ 1 test pour $j = 0$
→ 4 tests pour $j = 1$
→ 2 tests pour $j = 2$
→ 3 tests pour $j = 3$
→ 0 test pour $j = 4$
→ 0 test pour $j = 5$

Le nombre de tests effectués est $1 + 4 + 2 + 3 + 0 + 0 = 10$ tests, ce qui est inférieur à $n_1 + n_2 = 12$.
La complexité semble donc être un $O(n_1 + n_2)$.

OCaml
```
let rec inver_tab_tries = function
    |(L,[]) -> 0
    |([],L) -> 0
    |(x1::q1,x2::q2) -> if x1 < x2 then inver_tab_tries (q1,x2::q2)
                                   else (list_length x1::q1) + inver_tab_tries (x1::q1,q2);;

inver2 ([4;7;12;16;19;21],[2;3;6;11;15;20]);;
```

Complexité : La complexité de cet algorithme est un $O(n_1 + n_2)$.

Preuve

En effet, à chaque comparaison effectuée :
- $\rightarrow$ soit c'est l'indice i qui augmente de 1 (si on n'a pas une inversion),
- $\rightarrow$ soit c'est l'indice j qui augmente de 1 si on trouvé a une inversion.

Sachant que $\begin{cases} 0 \leq i \leq n_1 - 1 \\ 0 \leq j \leq n_2 - 1 \end{cases}$, i et j auront atteint leur valeur maximale après $n_1 + n_2 = n$ comparaisons.

2) Complexité de notre algorithme de décompte d'inversions

- Taille : n la taille du tableau traité.
- Unité de coût : la comparaison

Questions

1. Etablir l'ordre de grandeur de $f(n)$ la complexité des traitements complémentaires
2. Déterminer la relation de récurrence vérifiée par la complexité $c(n)$
3. En déduire que $c(n) = O(n \ln^2 n)$.

3) Programmation de l'algorithme

Questions

1. Faire la liste de toutes les sous-fonctions nécessaires accompagnées de leur fiche signalétique
2. Donner une implémentation de chacune de ces fonctions
3. En déduire la fonction de décompte du nombre d'inversions.